Data Tainting and Obfuscation: Improving Plausibility of Incorrect Taint

Sandrine Blazy, Stéphanie Riaud, Thomas Sirvent

Data Tainting and Obfuscation: Improving Plausibility of Incorrect Taint

Sandrine Blazy
Université de Rennes 1
IRISA
Email: Sandrine.Blazy@irisa.fr

Stéphanie Riaud
DGA Maîtrise de l'Information
Inria
Email: Stephanie.Riaud@inria.fr

Thomas Sirvent
DGA Maîtrise de l'Information
IRISA
Email: Thomas.Sirvent@m4x.org

***Abstract*—Code obfuscation is designed to impede the reverse engineering of a binary software. Dynamic data tainting is an analysis technique used to identify dependencies between data in a software. Performing dynamic data tainting on obfuscated software usually yields hard to exploit results, due to over-tainted data. Such results are clearly identifiable as useless: an attacker will immediately discard them and opt for an alternative tool.**

In this paper, we present a code transformation technique meant to prevent the identification of useless results: a few lines of code are inserted in the obfuscated software, so that the results obtained by the dynamic data tainting approach appear acceptable. These results remain however wrong and lead an attacker to waste enough time and resources trying to analyze incorrect data dependencies, so that he will usually decide to use less automated and advanced analysis techniques, and maybe give up reverse engineering the current binary software. This improves the security of the software against malicious analysis.

I. Introduction

Reverse code engineering consists in trying to understand the internal behavior of a binary program. This program, or a part of it, is analyzed to extract some relevant information from it. In some cases, this analysis is performed for malicious purpose, such as bypassing software protections or extracting some valuable algorithm from a compiled file. Protecting binary code from reverse engineering may therefore be crucial to avoid such threats.

Dynamic data tainting is an efficient analysis technique, used in reverse engineering, to follow data dependencies, i.e. the information flow (also called data flow), in the execution of a program. More precisely, when a variable value is modified, this information flow describes all data which are accordingly modified. Dynamic data tainting (more generally dynamic flow analysis) is used to find vulnerabilities in a binary code, to test programs (improve test coverage) [1], [2], [3], or debug programs (detect known bugs, such as null pointer dereferences) [4], [5], [6].

A dynamic data tainting analysis of a program, for a given set of interesting variables, consists in following precisely the impact of each of these variables, step by step, on all data manipulated in the execution of the program. Each data depending on some interesting variable is tainted and the concrete result of the analysis is a summary of specific properties checked on these tainted data. These properties are defined in a taint policy, and the goal of the analysis is to check precisely these properties. An execution trace may be given as an extra output, showing all steps concerning at least a tainted data (for example, a register or a memory chunk).

Data tainting is performed by automatic tools that implement a taint policy, defining precisely when a new taint is introduced, how to propagate a taint, and possibly what checks are performed on tainted data [7]. A new taint is usually introduced for input data, i.e. , data given to the program by the user. The taints are propagated through specific rules, corresponding to each possible instruction. The checks on tainted data are made either manually [8] or automatically [9] to detect specific behavior (for example, to check for a tainted particular test value in test coverage, or to detect tainted jump target addresses).

Consider as an example a software developer looking for vulnerabilities in his own program by performing a dynamic data tainting. Here, the dynamic data tainting analysis is to check that addresses used in jump instructions do not depend on values controlled by the user of the program (keyboard entries). In this case, the taint policy may be defined as follows. The taint is introduced for all keyboard input values. A simple propagation rule of the tainting tool is the following: when a tainted value is stored in a register and moved to another register, the taint is propagated to the destination register and the corresponding instruction is added to the execution trace. Once the program is executed, the developer checks in the resulting trace that no address is used as a tainted jump target in a jump instruction. Indeed, if such an address was tainted, then it could potentially be controlled from the keyboard by an attacker and the program would be vulnerable.

Defining an accurate taint policy is however tricky. Basic propagation rules do not take into account some specific cases where data seem independent from each other, while there are not. In such a case, the result is under-tainted and the analysis is not sound: some information related to tainted data is missing. To avoid this behavior, a cautious approach consists in tainting data, according to their potential dependency, even if their dependency is unclear. In this case, the analysis may lack of precision and give over-tainted results, where tainted data do not really belong to the data flow of the initially marked data. Such a taint analysis then results in huge execution traces that do not bring interesting information [10]. For example, a too permissive taint analysis could taint all cells of an array,

when only one cell of this array should be tainted during the considered program execution.

From another perspective, code obfuscation aims at transforming a program into a semantically equivalent program, such that a potential attacker needs much more effort to extract relevant information from the resulting program. The attacker (who aims at getting as much information related to the program as possible) will spend much more time and resources (for example, memory space) to extract relevant information from an obfuscated code.

Many obfuscation techniques exist in the literature [11]. Some of them, called data obfuscations, aim at making the data flow more complex and difficult to interpret. When they are efficient, these obfuscations significantly alter the results of taint analysis by suggesting fake dependencies between data: the analysis gives a strongly over-tainted result, impossible to exploit by an attacker.

This result is however useful for the attacker, who observes a proportion of tainted data that is much higher than expected and quickly realizes that the result is over-tainted due to an obfuscation-based software protection. After that, the attacker usually decides to use less automated and advanced analysis techniques (for example, switch to the IDA disassembler to visualize the control-flow graph and execute symbolically the program).

An original and interesting way of deluding the attacker is to modify the results of taint analysis so that the execution traces look plausible. In this situation, the attacker will exhaust available time and resources before giving up the taint analysis approach. A first way to improve the credibility of the taint analysis is to deliver reasonable rates for tainted data and tainted instructions.

This paper shows that the results of a taint analysis can be controlled, adding small code snippets to the program after the data obfuscation. These additions slightly transform obfuscated programs to reduce significantly the over-tainted results due to the obfuscation. In other words, our approach removes a lot of tainted data, so that the rate of tainted data and instructions are similar to the ones of the non-obfuscated program. The results of the data tainting analysis seem correct while they are still incorrect.

To achieve this goal, we have studied different state-of-the-art dynamic data tainting tools and the propagation rules of their taint policies. These propagation rules are meant to balance under-taint and over-taint and do not capture specific cases of data dependency. Our approach exploits these specific cases, by inserting code snippets in the program: these code snippets do not change the behavior of the program but alter the taint analysis (by removing taint). When inserted in strategic program points, these code snippets alter significantly the taint analysis so that the final trace becomes realistic. An attacker will then identify fake interactions between data, generated by the data obfuscation and simultaneously miss real interactions, hidden by our code snippets. The combination of data obfuscation with our code snippets will then strongly distort the understanding of the program.

II. Related Work

Data tainting is a well-known technique for finding bugs or vulnerabilities in software. For example, Feist *et al.* use data tainting to detect some vulnerabilities called use-after-free in [5] while Miller *et al.* detect bugs involving *null pointer dereferences* in [6].

Data tainting may be performed statically (i.e., without executing the code). The results are then valid for any execution of the program and thus may remain imprecise as they do not take into account specific input values and execution paths. Chang *et al.* [12] propose a static taint analysis based on the inputs given by the user. Ceara *et al.* improve this approach [13], by analyzing the control flow to improve the monitoring of data dependencies.

Dynamic program analysis offers another look at programs, studying their executions with various applications. Instrumentation frameworks for building dynamic analysis tools, like Pin [14] and Valgrind [15], provide an infrastructure for code profiling, bug detection, or performance evaluation.

Dynamic data tainting consists then in following data dependencies during an execution of a program. Dynamic data tainting is much more commonly used than static data tainting, in spite of its lack of generality: the result obtained is indeed easier to obtain and valid for a single execution [16]. A comprehensive survey of dynamic data tainting is proposed by Schwartz *et al.* in [7]. There are many applications of this technique to software security, such as preventing code injection attacks [9], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] or finding bugs in software [6], [17], [20], [27], [28], [29].

Among state-of-the-art dynamic taint analysis tools, TaintCheck [17] was designed to automatically detect standard overwrite attacks (exploiting well-known vulnerabilities, such as buffer overflow and format string). TaintCheck was the first tool to perform a dynamic taint analysis without requiring to compile or modify the source program. Dytan [9] was designed a few years later to introduce data taint at any memory chunk (and not only on network sockets as with TaintCheck). A more recent tool is TEMU [8], the dynamic analysis component of the BitBlaze tool suite [30]. TEMU is an extensible platform for fine-grained dynamic binary analysis. It performs analysis independently of the execution environment, thus facilitating more complex analyses and enabling more trustworthy results.

Even if dynamic data tainting tools have become very popular, their ability to analyze information flows remains limited: recent works [31], [32], [33], [34] present malware implementing techniques preventing efficient data taint analysis. These malwares are obfuscated, so that their control flow and data flow are much more complex to reverse engineer. Used techniques, consisting in control-flow obfuscations, aim at hiding control dependencies and explicit data assignments and lead to under-tainted results. In other words, the main idea in these works is to use control-flow based obfuscation techniques in order to make dynamic data taint analysis useless. Our work is different: we use data obfuscation techniques to

protect a binary software and then we modify the obfuscated binary software in order to make dynamic data taint analysis possible. Moreover, recent advanced data tainting tools are improving their taint propagation policy to preserve their ability to analyze such malwares. The main idea here is to add data tainting rules in order to track more data-flow and control-flow dependencies. The results are however mitigated: the obtained taint analyses contain indeed a lot of false positive data dependencies, and are thus over-tainted [33]. In the end, even if this approach gives interesting results on specific malwares, it does not lead to taint policies implementing very precise and scalable analyses. Our work takes advantage of this weakness and proposes code snippets exploiting the inherent lack of precision of dynamic taint analysis.

In a similar way, some low-level data obfuscation techniques obstruct dynamic data tainting analysis. These techniques add indirection levels to hide sensitive data and cause over-tainted results [31], [35]. In this last case, an attacker can hardly refine the taint policy to get more relevant results: these state-of-the-art data obfuscation techniques prevent taint analysis but the attacker can easily detect this behavior, as these obfuscation generate lots of over-tainted data. Restraining the over-tainting resulting from the dynamic data taint analysis would lead the reverse engineer to loose time analyzing fake results instead of switching to other reverse engineering techniques.

III. Our Contribution

This paper presents an approach and a tool for modifying the results of dynamic data tainting analysis. The goal of the modification is to generate plausible results of dynamic data tainting analysis on a program protected by a data obfuscation.

The notion of plausibility is difficult to quantify: multiple criteria may be used. We consider here that a result is plausible if the proportion of tainted instructions is roughly the same as the one of a similar unprotected program. This proportion corresponds indeed to the size of the execution trace given by the data tainting tool and is therefore the main indicator of under-taint or over-taint.

We study first different data tainting tools and their taint propagation policies. We define some representative propagation rules. As none of these tools was efficient and flexible enough (in terms of changes in the taint propagation policies), we use our own dynamic data tainting tool based on the Pin framework [14]; its taint propagation policy is typical of the policies we encounter in other tools.

Following [7], we observe that typical taint propagation rules cannot take into account specific behaviors of programs. From these specific behaviors, we define strategies to under-taint data and thus to mitigate the over-tainting effect of the data obfuscation. We confirm our approach with an experimental evaluation.

The remainder of this paper is organized as follows. Section IV details the taint propagation rules and explain why these rules can not be perfectly accurate. Section V shows how code snippets can remove taint from data and presents two examples of code snippets. Section VI details our experimentation process and introduces the tools we used. Section VII explains then our low-level code transformation, aiming at disrupting taint analysis. Section VIII describes lastly the experimental evaluation of our approach, followed by our concluding remarks.

IV. Taint Propagation Rules

In a taint policy, the introduction of new taints and the checks performed on tainted data are context-dependent. These parts are almost immediately derived from the target of the dynamic data tainting analysis (e.g. , debugging or finding vulnerabilities). On the other hand, taint propagation does not depend on the context and is quite difficult to define.

Taint propagation is based on rules, called propagation rules. Most data tainting tools use the same basic propagation rules. Beyond these common rules, each tool refines its analysis, by proposing specific rules (sometimes optional). We consider here fine-grained tools, where these propagation rules focus on instructions.

The general principle of taint propagation is summarized by Schwartz *et al.* [7]: *"The result of a binary operation is tainted if either operand is tainted, an assigned variable is tainted if the right-hand side value is tainted, and so on"*.

Rules existing in available data tainting tools (TEMU [8], TaintCheck [17], Dytan [9]) follow this general principle. The set of possible instructions is divided into three categories: data movement instructions (e.g., LOAD, STORE, MOVE, PUSH, POP), arithmetic instructions (e.g., ADD, SUB, XOR) and other instructions (e.g., NOP, JMP).

- Data movement instructions: the destination is tainted if the source is tainted.
- Arithmetic instructions: the result is tainted if any byte of the operands is tainted.
- Other instructions: no taint propagation occurs.

We give as an example a dynamic data tainting analysis of the C program described in Fig. 1, following the rules described above. We consider that the analysis tries to find out if the return statement depends on the input value.

```
x := getinput(·)
y : = 3 * x
return y
```

Fig. 1. Excerpt of a program for dynamic taint analysis

The executing program receives first an input value and stores it in the x variable: the x variable is therefore tainted. The value of the x variable is then multiplied by 3 and the result is stored in the y variable: the y variable becomes tainted, due to the arithmetic instruction with tainted operand x. In the end, the check detects that the return statement (and thus the variable y) is tainted.

With this small example, the rules seem particularly suitable. But dealing with real code is much more tricky. There are many cases where the basic rules described above are

not precise enough [7]. There are ways to move a piece of data without data movement instructions: the basic rules may then produce under-tainted results (e.g., because of control-flow dependencies). Arithmetic instructions do not mix data as much as expected by the basic rules, giving rise to over-taint. Some of these cases may be corrected using specific rules. Doing so has however an impact on the whole result of the data tainting analysis, because these specific rules apply on the whole execution of the program.

As a result, taint propagation rules are a subtle balance between rules trying to catch any dependency between data (to prevent under-taint) and rules trying to restrict the propagation to the cases where an explicit dependency occurs (to prevent over-taint).

To illustrate this behavior, we consider TEMU [8], a recent and state-of-the-art tool among publicly available dynamic data tainting tools [30]. TEMU is based on a secure environment (i.e., a virtual machine), performs a dynamic analysis of a binary code, and generates an execution trace using a plugin called TraceCap. It uses the basic rules described above and other rules defined for specific operations. For example, a specific rule concerns table lookups: in a table lookup, if a tainted input is used as an index to access an entry of the table, then the taint is not propagated to the destination. This specific rule (like other specific rules) can be activated or not. In fact, even if it is very effective in operations such as conversions between Unicode and ASCII in Windows, this specific rule tends to under-taint the results [7], which shows that it is difficult to develop rules that match the semantics of each program.

V. Taint Adjusting Code Snippets

Data obfuscation leads to over-tainting results in dynamic data tainting, with basic taint propagation rules. Our goal is to reduce significantly the over-taint so that the results remain plausible. We indentify cases where basic propagation rules offer an opportunity to remove taint [7]. We give as examples code snippets taking advantage of this opportunity by removing taint on some data stored in registers.

The concrete insertion of these code snippets and their locations are described in Section VII. We show however here that many code snippets can be built by a careful study of basic propagation rules. We show moreover that these code snippets are small and can be written with a lot of variety, so that they remain stealth when inserted in programs.

The dynamic nature of the targeted data tainting analysis offers a first opportunity. Dynamic tools have a precise knowledge of all executed instructions but have no clear understanding of the control flow. We consider a conditional jump instruction: the branches created by this instruction manipulate data in different ways. If the condition used in the jump instruction depends on tainted data, then all data manipulated in the branches depend on this tainted data as well: they should therefore be tainted. Yet, because a dynamic analysis has no way to detect when the branches end, such taint propagation cannot be realized in practice (i.e., there is no propagation rule corresponding to this situation).

Using this lack of propagation, the code snippet given in Fig. 2 will remove the taint of the register `ebx` supposed to be tainted. This code snippet requires that registers `eax` and `ecx` are available.

```
      pushf
      mov  eax, 0
      mov  ecx, 1
l1:   test ebx, ecx
      jz   l2
      xor  eax, ecx
l2:   shl  ecx, 1
      jnz  l1
      mov ebx, eax
      popf
```

Fig. 2. An example of code snippet influencing the taint propagation

More precisely, the register `eax` is initialized with value 0. Then, during the execution of a loop, the value of each bit of `ebx` is tested (using `ecx` to select the appropriate bit). If the value is 1, then the bit is added at the same position in `eax`. In this code snippet, the register `ecx` is never tainted, because it is initialized with a constant, and then simply shifted. The register `eax` is not tainted either, because it is initialized with a constant and then modified with register `ecx`. In the last `mov` instruction, the register `ebx` receives the value of the register `eax`: its taint is therefore removed, even if the register `eax` contains in fact the exact same value as `ebx`.

Another opportunity comes from flags. Flags are modified by many instructions according to complex rules and are therefore ignored by taint propagation rules. A convenient way of removing taint is to use a flag as a temporary storage. Using this principle, the code snippet given in Fig. 3 will remove the taint of the register `ebx`, supposed to be tainted, using instruction `lahf` to read flags. This code snippet requires that registers `eax`, `ecx`, and `edx` are available.

```
      pushf
      mov  eax, 0
      mov  ecx, 32
      mov  edx, 0
l1:   shl  ebx, 1
      shl  edx, 1
      lahf
      and  eax, 1
      xor  edx, eax
      sub  ecx, 1
      jnz  l1
      mov  ebx, edx
      popf
```

Fig. 3. Another example of code snippet influencing the taint propagation

More precisely, the register `edx` is initialized with value 0. Then, the value of each bit of `ebx` is moved to the carry flag (using a left shift instruction). This flag value is captured in `ah` (and thus `eax`) with the `lahf` instruction and then inserted

in register edx. With the left shifts applied to registers ebx and edx, the register edx will receive the initial value of register ebx. In this code snippet, the register eax is never tainted, because it is initialized with a constant and modified by flags and a constant only. The register edx is not tainted either, because it is initialized with a constant, shifted, and modified with register eax. In the last mov instruction, the register ebx receives the value of the register edx: its taint is therefore removed, even if it receives in fact its initial value.

This example may look similar to the previous one. The conditional jump here is however only used for the 32 iterations of the loop. We could have written a different (larger) code snippet without any conditional jump. In this work, we have used only two patterns of code snippets (and their multiple variants, see section VII). As future work, we intend to define more patterns of code snippets from propagation rules.

VI. Experiments

This section describes our approach for testing the impact of data obfuscation on dynamic data tainting analysis. Its four steps are shown in Fig. 4. In the following, we consider that all the software tools are executed on an x86 architecture.

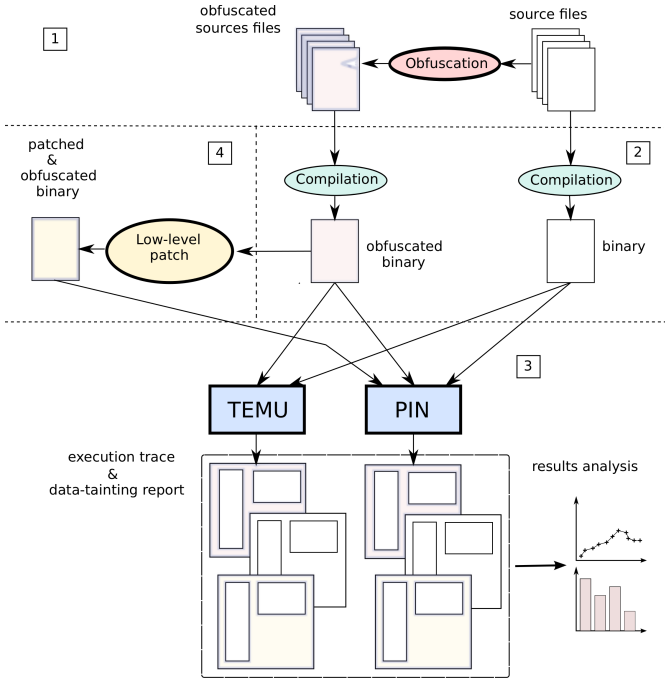

Fig. 4. Testing the impact of a data obfuscation on a data tainting analysis

Firstly, a source program is obfuscated. Secondly, the initial program and the obfuscated programs are compiled. Thirdly, a dynamic data tainting analysis is performed on each of compiled programs. Each data tainting analysis is performed twice: once with our own tool and once with the TEMU tool that we chose as reference. Fourthly, our two code snippets are embedded in binary code to ensure that the dynamic data tainting analysis of obfuscated code gives plausible results that the attacker can attempt to exploit during reverse engineering. This last step will be explained in the next section.

Following the criteria defined by Cavallaro *et al.* [31], we chose a data obfuscation able to defeat dynamic data tainting analysis. This data obfuscation operates over C programs. It obfuscates the data flow of a program by adding indirection levels (mainly by dereferencing pointers and function calls) in the access of some fields of records, and also by randomly modifying their addresses in memory at each field access [36].

A dynamic data tainting analysis is performed, after compilation, on both programs (original and obfuscated). The results are compared together: their difference measures the impact of the data obfuscation on the dynamic data tainting analysis. In this third step, we use two dynamic tainting tools: TEMU [30] and our own tool based on the Pin framework [14].

TEMU does not support taint input from a memory region corresponding to a data structure, which is not convenient for tainting initially our obfuscated programs. We modify our source programs to introduce taint data from the keyboard and use the TraceCap plug-in for the taint analysis with default propagation rules. At the end of the dynamic data tainting analysis, TEMU generates a report and an execution trace. Fig. 5 shows an excerpt of such a report. It gives the number of instructions that were decoded from the binary file, the total number of decoded operands (each instruction usually consists of 0 to 2 operands), and the size of the execution trace. The example given in Fig. 5 comes from a 10 gigabytes execution trace. The number of decoded instructions is 4.5 times the number of tainted instructions, which is a representative number in our experiments.

```
Stop tracing process 2844
Number of instructions decoded: 315585949
Number of operands decoded: 744026854
Number of tainted instructions written to trace: 72449628
Processing time: 1560.6 U: 1548.64 S:11.9602
```

Fig. 5. Excerpt of a taint analysis report generated by TEMU

The execution trace contains a sequence of the executed instructions, including the values of their operands and tainted data resulting from the taint propagation. We use the Vine plug-in of the BitBlaze tool suite [30] to get a readable view of such execution trace.

TEMU is a sophisticated tool that gives precise and exhaustive results related to the taint propagation. Indeed, TEMU is monitoring and tracking the data of a binary file from a virtual machine including standard libraries and other processes. The size of the collected traces is however huge (up to a few dozens of gigabytes) and their interpretation is therefore difficult.

The main cause for these huge traces is that TEMU is monitoring too many events, including all the interactions with the runtime system. Even if this approach is generally considered as a real strength of TEMU, we only need in our experiments to instrument the instructions corresponding to the C statements from the original source file that we obfuscated. In our experiments with TEMU, we managed to extract general

information related to the behavior of the taint analysis (e.g., increase of the number of tainted instructions).

To understand more precisely the behavior of the taint analysis together with the impact of our data obfuscation on the over-taint, we chose however to implement our own dynamic data tainting analysis tool using the Pin generic framework [14] with similar propagation rules. Such a tool is called a PinTool; it may add some instructions in the currently analyzed code: the resulting code is then executed and interpreted.

Our PinTool performs a dynamic data tainting analysis, based a taint policy similar to the one used by TEMU. The taint is initialized and propagated on memory addresses and registers. First, we use a debugger to select the memory addresses corresponding to the initially tainted data and also the range of addresses of instructions that we trace (this range of addresses is used to restrict the dynamic data tainting analysis to a specific part of a program, such as a function or a loop). Then, we instrument the code, instruction by instruction, to propagate the taint: a memory address or a register is tainted when it stores a tainted value. Our PinTool builds a report and an execution trace, similar to the results given by TEMU.

Data obfuscation is applied on specific sensible data. Our PinTool initially taints the memory addresses and registers containing the data that the obfuscation tries to protect. The taint propagation rules are similar to the ones of TEMU, even if these tools are very different from each other. Indeed, TEMU is a very effective tool, it generates very precise execution traces containing information of which we do not take advantage (e.g., the states of the register flags) in our working context. Our PinTool generates much shorter execution traces than TEMU and these traces can therefore be more easily analyzed.

```
Taint source : 6500196 (0x632F64)
Start address : 4215184
End   address : 4346821
Number of executed instructions : 702456
Number of tainted instructions : 976
List of Tainted addresses : [632f64] [632f96]
List of Tainted registers : RAX
#eof
```

Fig. 6. Excerpt of a taint report generated by our PinTool

Fig. 6 shows an excerpt of a taint report, given by our PinTool. This taint report contains the memory address of the initial taint (called taint source) and the range of addresses of traced instructions `[start address; end address]`. We can adjust this range, for example to get shorter execution traces. It also indicates information that was not mentioned in the TEMU report: the list of tainted addresses and registers at the end of the execution.

Fig. 7 shows an excerpt of an execution trace given by our PinTool. This execution trace consists in general information related to each instruction (its address, its disassembled corresponding instruction, accessed registers and access modes) and information resulting from the taint propagation.

For example, in Fig. 7, the fourth instruction (first instruction stored at address `4063a8`) is a `mov` instruction; its source is tainted, so its destination, the `eax` register, becomes tainted after the execution of this instruction. The next instruction in the trace is exactly the same instruction, located at the same address in the code: this may happen for instructions placed in loops. Its execution does not modify the taint of the register. After the last instruction, the register is not tainted anymore and thus removed from the list of tainted registers.

```
[WRITE in 6331ec] 405992: mov dword ptr [rdx], ecx
> 6331ec tainted
    (...)
    (...)
[WRITE in 632f28] 405231: mov dword ptr [rbx+0xfb48], 0x0
[WRITE in 632f28] 405231: mov dword ptr [rbx+0xfb48], 0x0
[READ  in 632f28] 4063a8: mov eax, dword ptr [rdx+0xfb48]
> eax tainted
[READ  in 632f28] 4063a8: mov eax, dword ptr [rdx+0xfb48]
> eax already tainted
[READ  in 632f00] 4062eb: mov eax, dword ptr [r12+0xfb18]
> eax freed
```

Fig. 7. Excerpt of an execution trace generated by our PinTool

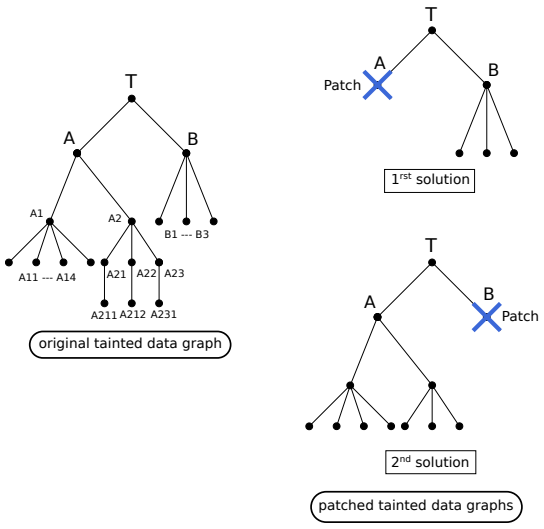

Fig. 8. Impact of a patch location on data tainting

VII. Adjusting Data Taint

Our goal is to hide the data obfuscation process from a dynamic data tainting analysis and provide a plausible taint to the attacker. We focus on the most simple indicator of plausibility, the proportion of tainted instruction given by the analysis. We consider that a result of dynamic data tainting analysis is plausible when the over-taint caused by the transformations applied to the code does not exceed 10%. This difference is measured between on the one hand the original code and on the other hand the modified (after the insertion of code snippets) and obfuscated code (obfuscated code with inserted code snippets).

This section details the transformations that we propose to reduce significantly the over-taint resulting from data obfus-

cations. More precisely, we insert code snippets, like the ones described in Section V, in the obfuscated code. We adapt these code snippets to the context of their insertion, so that they remove the taint from specific sensible registers (containing the sensible data, or strongly related data). These code snippets stop then the taint propagation.

The location of these code snippets is of primary interest. Due to their usage, we call them stop points. If they are reached too late during program execution, then the over-taint remains and an attacker will realize that software protections are used in the code. On the other hand, if they are reached too early, not enough data will be tainted, thus making the dynamic data tainting analysis useless, since the result would be recognizable as under-tainted.

Thus, these code snippets must be located at strategic program points to achieve an optimal impact on the result of the dynamic data tainting analysis. They should be placed closely before points where the taint is spread from a single data to multiple data. This is typically the case of function calls with a tainted parameter (or recursive functions involving tainted data). These code snippets have moreover an impact only if they are concretely executed, during the dynamic data tainting analysis. Their location must be chosen according to the coverage of dynamic executions. At least, we must assure that these code snippets are reachable (a single dynamic trace is enough). We try then to select locations that are reached by almost all dynamic executions of the program. We consider at last parallel branches to place a code snippet on each branch, so that no execution will bypass all code snippets.

The target of the code snippets, among sensible data, is important as well. The right of Fig. 8 shows two graphs resulting from dynamic data tainting performed after the application of a code snippet. They represent two possible solutions for stopping the dynamic data tainting. In this figure, we only show the nodes representing tainted data for a specific execution and consequently for a possible path. The first solution chooses node A as a target and removes the taint of this node closely before its propagation. This solution has more impact on data tainting than the second solution (which chooses node B). Indeed, the size of the subgraph having node A as root is larger than the size of the other candidate subgraph. Thus, our strategy is to choose the first solution, which reduces the most the number of tainted nodes in the graph. In practice, a first dynamic data tainting analysis is performed before the application of code snippets, to detect taint dependencies, and find accurate targets for code snippets.

Our solution concretely consists in adding code snippets in obfuscated assembly code. The stealth of these code snippets is crucial: if we insert too many similar snippets in a code, or if our code snippets consist of too many instructions, it will be easy for an attacker to recognize them in an obfuscated binary code. To avoid such situations, we reduce the sizes of our code snippets as much as possible. The negligible size of these code snippets and the fact that we can decline them into a multitude of different versions, makes them less easy to detect. To generate different versions of a code snippet, we can for instance expand loops, or use equivalent instructions (use "`xor eax, eax`" instead of "`mov eax, 0`", ...).

Our process for transforming binary files is illustrated in Fig. 9. Firstly, we determine the memory addresses of the stop points. To that purpose, we compile our source files and use the IDA Pro tool [37] to disassemble the generated binary files and to select precisely the stop points, as discussed previously. Secondly, because we know where to add our code snippets, we embed inline assembly code in our source files. Thirdly, we compile these source files once again and reuse IDA Pro to adjust, by modifying, the embedded code snippets.

The registers used in our assembly code snippets are fixed and we may have to change them depending on the results of register allocation. If a register used by a code snippet is also used by the program, then we must change it (into a free register) in the code snippet to avoid interferences between the code snippet and the original program. Renaming registers in code snippets is performed in a fourth step, using IDA Pro. In doing so, we are sure that our code snippets do not change the semantics of the analyzed program.

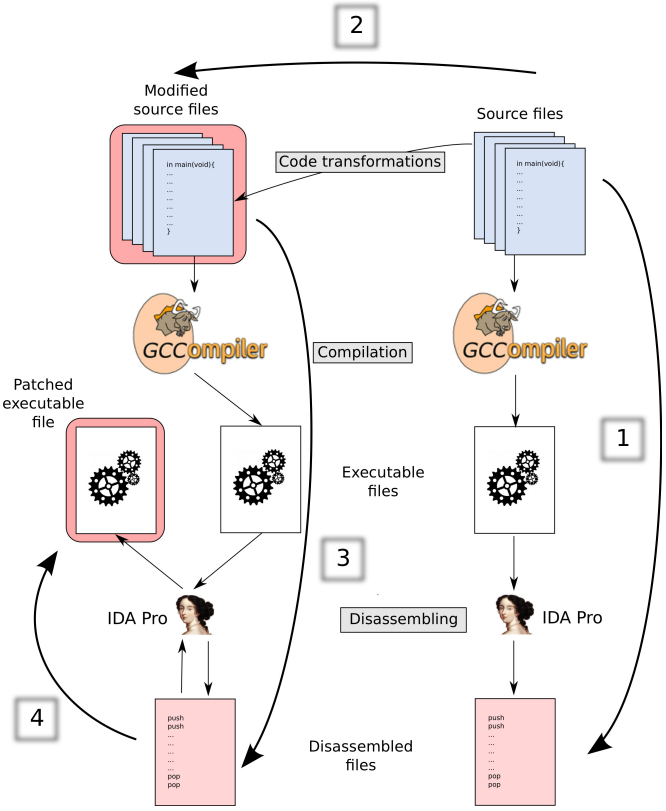

Fig. 9. Transforming code in order to modify the results of dynamic data tainting analysis

VIII. Experimental Results

This section details our experimental results obtained after obfuscating C source programs and performing dynamic data tainting of resulting binary files.

A. Selecting a Test Program

As our obfuscation hides some selected fields of data in C structures (`struct`), we consider C programs defining structures such that some of their fields are significantly updated during program execution. After testing our approach on small C programs, we selected a real program, the calculator integrated into the GNOME desktop environment. This software consists of about 14,000 lines of C code, written in 32 source files. These files share a common data structure manipulating some fields that could be worth considering as sensitive data (e.g., because they are involved in the precision of the final result given by the calculator). Follwing [36], we choose to obfuscate four integer fields belonging to this data structure and deemed to be sensitive fields. Then, we check that accesses to these fields are performed during program execution.

Then, we follow the process described in Fig. 4 and compile the original and obfuscated programs using GCC version 4.8.0 [38] at the first level of optimization. The last step of the process is the insertion of code snippets in binary code.

In the calculator code, we choose a stop point corresponding to a function call. This function takes as argument one of the initially tainted fields and introduces a lot of taint propagation. To stop this taint propagation, we insert a code snippet at the beginning of the code of the function. In this example, choosing a single stop point is sufficient to generate a plausible taint.

B. Analyzing Tainted Programs

We define several execution scenarios that we follow for each of our taint analyses. They are representative of standard executions of the calculator software.

Concerning the taint analyses performed by TEMU, we obtain execution traces varying from 10 gigabytes (for non-obfuscated programs) to 30 gigabytes (for obfuscated programs). Such traces are too big to be precisely analyzed. The tainting reports indicate that about 200 millions of instructions were decoded by TEMU in the non-obfuscated program, including about 70 millions of tainted instructions. In the obfuscated code, more than one billion of instructions are decoded, including more than 400 millions of tainted instructions. The proportion of tainted instructions with respect to executed instructions thus increases by 21% when obfuscating the source program.

Concerning the dynamic data tainting analyses performed by our PinTool, about 700,000 instructions are instrumented in the non-obfuscated and obfuscated versions of the program. The precise average numbers are given in Table I. Among these instructions, 3,392 are tainted in the non-obfuscated program and 4,688 are tainted in the obfuscated program. The proportion of tainted instructions with respect to executed instructions thus increases by 37% when obfuscating the source program (the number of tainted instructions increases by 38% from the non-obfuscated code to the obfuscated one but the number of executed instructions increases by only 0.6%).

With both dynamic data tainting tools, we observe that our C data obfuscation has increased the number of tainted instructions. The number of executed instructions is much smaller with our PinTool as it only executes the instructions starting at the beginning of the main function of source programs and ending when the main function yields a result. Similarly, the number of tainted instructions is much bigger with TEMU because we had to change the source program to introduce the initial data from the keyboard (and not directly at a memory location as we did with our PinTool). As a consequence, the interactions with the keyboard are also taken into account in the dynamic data tainting analysis. Lastly, we also observe that the number of executed instructions has increased when the program is obfuscated.

C. Transforming Binary Code

The code snippet that we add in the source program as an inline assembly is our first example described in Section V and mentioned in Fig. 2. It is applied on a tainted register (`r13d` in our experiment, instead of `ebx`), containing the value of an obfuscated integer field, and located in a loop of a recursive function of the program. As soon as the two temporary registers used in this code snippet (`eax` and `ecx`) are available at the program point where we insert the code snippet, this sequence of instructions does not interfere with the original code. If one of these registers is not available, then we need to use an available register.

As shown in Fig. 10, the taint of register `r13d` is removed by the code snippet, which stops the taint propagation after the last instruction of the code snippet we added in the program.

```
[WRITE  in R13D]   40cb38: mov r13d, eax
> r13d freed
```

Fig. 10. Excerpt of an execution trace, where taint of `r13d` is removed

D. Results of the Patched Taint Analysis

The average results with our PinTool are summarized in Fig. 11 and Table I. Our data obfuscation adds only 46 lines of C instructions. After performing obfuscation and applying assembly code snippets, the number of executed instructions increases slightly; the number of tainted instructions decreases from 4,688 to 3,560 tainted instructions.

In the obfuscated and patched programs, the proportion of tainted instructions is similar to the proportion of tainted instructions in the non-obfuscated original program. Indeed, the difference between both codes is only 4%. Furthermore, the code snippet has decreased the proportion of tainted instructions by 24% on the obfuscated program. The low increase in executed instructions results from the instructions of the code snippet in the program.

Our results show that our low-level transformations have the expected influence on dynamic data tainting analysis. We could even add other code snippets (i.e., that would be syntactically different but semantically equivalent to the code snippet of Fig. 2) at other stop points to decrease even more the

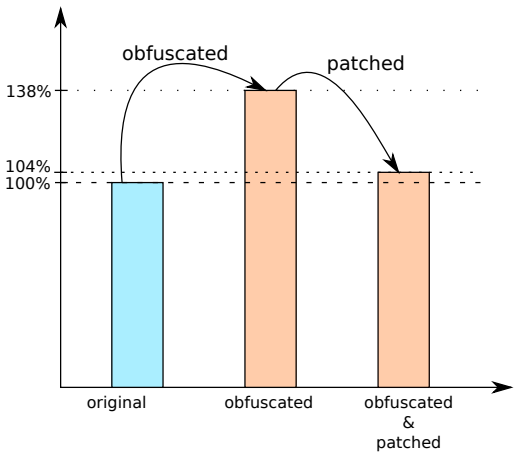

Fig. 11. Proportion of tainted instructions (with respect to original file)

number of tainted data. Thus, we generated a plausible taint resulting from simple and small code changes. Diversifying the code snippets is also an interesting solution for generating furtive changes in binary code.

E. Summary of the Experiments

The insertion of a single code snippet restored the number of tainted instructions, after data obfuscation, on an executable generated from more than 10,000 lines of code. The number of code snippets to be inserted remains thus relatively low if their targets and locations are carefully chosen. Moreover, this kind of protection can be inserted by someone having a good knowledge of the program, in a reasonable time (depending on the size of the source code, and the skills of the programmer): a few hours were necessary to identify the target and insert the code snippet in the calculator. This technique is then currently manual, and requires some expertise, but may be improved to become automated during the compilation, like an optimization. In addition, the number of deployed code snippets must remain low not to stop the propagation of the taint too much, which would alert the attacker.

More generally, we have defined assembly code snippets aiming at bypassing taint propagation rules that are commonly used by dynamic data tainting tools. The dynamic data tainting tools could evolve and improve their taint propagation policy to propagate data tainting even in the presence of code snippets.

Such adaptations of the taint propagation policy are however not easy to build. Some instructions, such as conditional jump instructions, are particularly tricky to handle. Too permissive propagation rules will result in lots of tainted data after these instructions, even with few data dependencies between these tainted data. Such propagation rules would then probably generate over-tainted data, even in non-obfuscated programs.

An alternative and more permissive approach would strive for avoiding an excessive over-taint and would necessarily lead to a trade-off between taint propagation and the definition of precise propagation rules. Such a permissive handling of some instructions would still leave the opportunity of building new code snippets for stopping the taint propagation.

TABLE I
SUMMARY OF AVERAGE RESULTS OBTAINED WITH OUR PINTOOL

	Original	Obfuscated	Obfuscated and Patched
Size of source program (in lines of C)	14 009	14 055	14 074
Size of original binary file (in KBytes)	403	406	406
Number of executed instructions	699633	703 713	713 471
Number of tainted instructions	3392	4688	3560
Number of additional tainted instructions	0	1292	168
Rate of increase of tainted instructions	0%	38%	4%

IX. CONCLUSION

Data obfuscation techniques tend to over-taint the results from dynamic data tainting, thus revealing the use of obfuscation techniques for generating a binary code. Attackers very often use automatic data tainting tools when reverse engineering programs, except when they realize that these tools yield too much over-tainted data.

We show that modifying assembly code with a small number of code snippets is an interesting way of fooling a dynamic taint analysis. Most efficient obfuscation techniques involve over-tainted data resulting from taint analysis. We are nevertheless able to mitigate the over-taint and thus to simulate a successful and plausible data tainting analysis. Facing our protection, an attacker will spend a lot of time in analyzing fake data. A more concrete evaluation of this protection, using multiple code snippets and target programs, should confirm our first experimental results.

Taking into account our patches in a taint propagation policy seems to be difficult. Looking for these code snippets in a binary code seems more reasonable. This is obviously even easier when a same code snippet is reused at different locations in a binary code. As future work, we shall study how to make the detection of these code snippets more complex. We intend to diversify our patches and to intertwine them with the rest of the code.

REFERENCES

[1] T. R. Leek, R. E. Brown, M. A. Zhivich, T. R. Leek, and R. E. Brown, "Coverage maximization using dynamic taint tracing," MIT Lincoln Laboratory, Tech. Rep. TR-1112, 2007.

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[4] W. Masri, A. Podgurski, and D. Leon, "Detecting and debugging insecure information flows," in *15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004.*, Nov 2004, pp. 198–209.

[5] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use-after-free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. online article, January 2014.

[6] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Crash analysis with BitBlaze," in *In BlackHat 2010, Las Vegas, NV, July 2010.*, 2010.

[7] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, 2010, pp. 317–331.

[8] H. Yin and D. Song, "TEMU: Binary code analysis via whole-system layered annotative execution," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3, Jan 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html

[9] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. New York, NY, USA: ACM, 2007, pp. 196–206. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273490

[10] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. New York, NY, USA: ACM, 2010, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831711

[11] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Sciences, The University of Auckland, Tech. Rep. 148, 1997.

[12] R. M. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, "Inputs of coma: Static detection of denial-of-service vulnerabilities." in *Computer Security Foundations (CSF)*. IEEE Computer Society, 2009, pp. 186–199.

[13] D. Ceara, L. Mounier, and M.-L. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '10)*. IEEE Computer Society, 2010, pp. 371–380.

[14] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," in *In Prooceeding of the third workshop on Runtime Verification (RV'03)*, 2003.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 190–200.

[16] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*. Citeseer, 2003, pp. 24–27.

[17] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[18] S. Heelan and A. Gianni, "Augmenting vulnerability analysis of binary code," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. New York, NY, USA: ACM, 2012, pp. 199–208.

[19] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *In Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 63–78.

[20] R. W. M. Jones, P. H. J. Kelly, M. C, and U. Errors, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *in Distributed Enterprise Applications. HP Labs Tech Report*, 1997, pp. 255–283.

[21] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, pp. 85–96, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1037947.1024404

[22] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 221–232. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2004.26

[23] W. Halfond, A. Orso, and P. Manolios, "Wasp: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 65–81, 2008.

[24] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 22–22.

[25] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[26] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135–148.

[27] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005.

[28] J. Richard, "Taint nobody got time for crash analysis," in *REcon : Computer Security Conference on Reverse Engineering and Advanced Exploitation Techniques*, Montreal, Quebec, CANADA, 2013.

[29] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 116–127.

[30] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security.*, Dec. 2008.

[31] L. Cavallaro, P. Saxena, and R. R. Sekar, "Anti-taint-analysis: Practical evasion techniques against information flow based malware defense," Stony Brook Computer Science Dept, Tech. Rep., 2007.

[32] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 5137, pp. 143–163.

[33] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli, "Protection against Code Obfuscation Attacks based on control dependencies in Android Systems," in *8th Workshop on Trustworthy Computing (TC'14)*, 2014.

[34] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. New York, NY, USA: ACM, 2010, pp. 13–24.

[35] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, Sept 2014, pp. 255–264.

[36] S. Blazy and S. Riaud, "Measuring the robustness of source program obfuscation: Studying the impact of compiler optimizations on the obfuscation of c programs," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY'14)*. New York, NY, USA: ACM, 2014, pp. 123–126.

[37] IDA, the multi-processor disassembler and debugger. https://www.hex-rays.com/products/ida/.

[38] GCC, the gnu compiler collection. https://gcc.gnu.org/.